Software Development (Algorithms and Data Structures) MOCK Examination - Solution

Prof. Dr. Dominik Böhler
Time Allowed: 90 Minutes
Total Points: 90 Points

Question 1: Fundamental Concepts and Linked Lists (30 points)

(a) Conceptual Understanding (10 points)
(i) Briefly explain the primary difference between a Stack and a Queue in terms of how elements are added and removed. (4 points)

* Solution:
* A Stack is a Last-In, First-Out (LIFO) data structure. Elements are added (pushed) and removed (popped) from the same end, typically called the "top" of the stack. The last element added is the first one to be removed.
* A Queue is a First-In, First-Out (FIFO) data structure. Elements are added (enqueued) at one end (the "rear" or "tail") and removed (dequeued) from the other end (the "front" or "head"). The first element added is the first one to be removed.

(ii) What is an **Abstract Data Type (ADT)**? Provide one example of an ADT and briefly describe its common operations. (6 points)

*   **Solution:**
    *   An **Abstract Data Type (ADT)** is a mathematical model for data types, defining the data stored and the operations that can be performed on that data, without specifying how the data is stored or how the operations are implemented. It focuses on *what* the data type does, rather than *how* it does it.

    *   **Example: Stack ADT**
        *   **Data:** A collection of elements, ordered in a LIFO manner.

*   **Common Operations:**
    *   `push(element)`: Adds an element to the top of the stack.
    *   `pop()`: Removes and returns the element from the top of the stack.
    *   `peek()`: Returns the element at the top of the stack without removing it.
    *   `isEmpty()`: Checks if the stack contains any elements.
    *   `size()`: Returns the number of elements in the stack.

(b) Linked List Operations (10 points, +4 Points in Rust)

Consider a singly linked list where each node contains an integer and a next pointer. Implement a function that inserts a new node with a value at the end of the linked list. Handle the case of an empty list.

Choose ONE of the following languages for your implementation:

(i) Python Implementation: (10 points)

(ii) Rust Implementation: (10 + 4 points)

(c) Array vs. Linked List (10 points)
Discuss two advantages of using a singly linked list over a dynamic array for storing a collection of elements, and two disadvantages. (10 points)

* Solution:
* Advantages of Singly Linked List over Dynamic Array:

1. Dynamic Size / Efficient Insertions/Deletions (at ends/middle): Linked lists can grow or shrink dynamically without needing to reallocate and copy elements, unlike dynamic arrays which may require resizing operations. Inserting or deleting elements in the middle of a linked list (once the position is found) is O(1) because it only involves changing a few pointers, whereas in a dynamic array, it requires shifting all subsequent elements, which is O(N).

2. No Wasted Memory (if elements are sparse): Linked lists only allocate memory for the nodes actually used, avoiding the potential for wasted space if a dynamic array is allocated with a larger capacity than currently needed.

*Disadvantages of Singly Linked List compared to Dynamic Array:**

1. **No Random Access:** Accessing an element at a specific index (e.g., the i-th element) in a linked list requires traversing from the beginning, resulting in O(N) time

complexity. Dynamic arrays provide O(1) random access using indices.

2. **Higher Memory Overhead:** Each node in a linked list requires extra memory for the `next` pointer (and potentially `prev` pointer in a doubly linked list) in addition to the data itself. Dynamic arrays store only the data, leading to better memory utilization for large collections of small elements.

Question 2: Trees and Hashing (30 points)

(a) Binary Search Tree (BST) Operations (15 points)
Given an initially empty Binary Search Tree (BST), insert the following sequence of integer keys in the given order: 45, 20, 70, 10, 30, 60, 80, 25.

(i) Draw the final structure of the BST after all insertions. (8 points)

* Solution:

```
            45
          /    \
      20              70
     / \             / \
   10    30        60    80
        /
      25
```

(ii) Perform an **in-order traversal** on the final BST and list the nodes in the order they are visited. (7 points) [3, 5]

* Solution:
  * In-order traversal visits nodes in the order: Left subtree, Root, Right subtree. For a BST, this results in elements being visited in ascending order.
  * Order: `10, 20, 25, 30, 45, 60, 70, 80`

(b) Hashing with Linear Probing (15 points)
Consider a hash table of size 7 (indices 0 to 6) and a hash function h(key) = key % 7.
Insert the following keys into the hash table using linear probing for collision resolution: 44, 45, 79, 55, 91, 18.
Draw the final state of the hash table, showing the key at each index. (15 points)

* Solution:
* Hash Table Size: 7
* Hash Function: h(key) = key % 7
* Keys to insert: 44, 45, 79, 55, 91, 18

1. **Insert 44:** `h(44) = 44 % 7 = 2`. Table[2] = 44.
   `[_, _, 44, _, _, _, _]`
2. **Insert 45:** `h(45) = 45 % 7 = 3`. Table[3] = 45.
   `[_, _, 44, 45, _, _, _]`
3. **Insert 79:** `h(79) = 79 % 7 = 2`. Collision at index 2.
   * Probe 1: Index 2 is occupied by 44.
   * Probe 2: Index 3 is occupied by 45.
   * Probe 3: Index 4 is empty. Table[4] = 79.
   `[_, _, 44, 45, 79, _, _]`
4. **Insert 55:** `h(55) = 55 % 7 = 6`. Table[6] = 55.
   `[_, _, 44, 45, 79, _, 55]`
5. **Insert 91:** `h(91) = 91 % 7 = 0`. Table[0] = 91.
   `[91, _, 44, 45, 79, _, 55]`
6. **Insert 18:** `h(18) = 18 % 7 = 4`. Collision at index 4.
   * Probe 1: Index 4 is occupied by 79.
   * Probe 2: Index 5 is empty. Table[5] = 18.
   `[91, _, 44, 45, 79, 18, 55]`

Final State of Hash Table:

Index	**Value**
0	91
1	
2	44
3	45
4	79
5	18
6	55

Question 3: Algorithmic Analysis and Graph Traversal (30 points)

(a) Asymptotic Analysis and Recurrence Relations (15 points)

(i) Determine the Big O time complexity for the following pseudocode snippet. Justify your answer. (7 points)
pseudocode Algorithm Example(N):

```
sum = 0
for i from 1 to N:
    for j from 1 to i:
        sum = sum + 1
```

* Solution:
* The outer loop runs N times (for i from 1 to N).
* The inner loop runs i times for each iteration of the outer loop.
* When i = 1, inner loop runs 1 time.
* When i = 2, inner loop runs 2 times.
* ...
* When i = N, inner loop runs N times.
* The total number of sum = sum + 1 operations is 1 + 2 + 3 +... + N.
* This sum is equal to N * (N + 1) / 2, which simplifies to (N^2 + N) / 2.
* In Big O notation, we take the highest order term and drop constants. Therefore, the time complexity is O(N^2)

(ii) Solve the following recurrence relation using the Master Theorem or by repeated substitution to find its asymptotic time complexity: `T(n) = 3T(n/3) + O(n)`. (8 points)

* **Solution (using Master Theorem):**
  * The Master Theorem applies to recurrences of the form `T(n) = aT(n/b) + f(n)`.
  * In this case, `a = 3`, `b = 3`, and `f(n) = O(n)`.
  * We need to compare `f(n)` with `n^(log_b a)`.
  * `log_b a = log_3 3 = 1`.
  * So, we compare `f(n) = O(n)` with `n^1 = n`.
  * This falls into **Case 2** of the Master Theorem, where `f(n) = Theta(n^(log_b a))`.
  * Therefore, the solution is `T(n) = Theta(n^(log_b a) * log n) = Theta(n * log n)`.
  * **Asymptotic Time Complexity: O(n log n)** [11, 5]

(b) Graph Traversal (15 points)

Consider the following undirected graph:
Nodes: A, B, C, D, E, F
Edges: (A,B), (A,C), (B,D), (C,E), (D,F), (E,F)

(i) Draw the graph. (5 points)
*   **Solution:**
```
A --- B
|     |
C --- D
|     |
E --- F
```
(Note: This is one possible drawing; connectivity is the key.)

(ii) Perform a **Breadth-First Search (BFS)** starting from node 'A'. List the nodes in the order they are visited. When multiple unvisited neighbors are available, visit them in alphabetical order. (10 points)

*   **Solution:**
    *   **Queue (Q):** Stores nodes to visit.
    *   **Visited Set (V):** Stores visited nodes.

    1. Start at 'A'.
        *   Q: [A]
        *   V: {A}
        *   Visited Order: A

    2. Dequeue 'A'. Neighbors of 'A': B, C. Add alphabetically.
        *   Q:
        *   V: {A, B, C}
        *   Visited Order: A, B, C

    3. Dequeue 'B'. Neighbors of 'B': A (visited), D. Add D.
        *   Q:
        *   V: {A, B, C, D}
        *   Visited Order: A, B, C, D

4. Dequeue 'C'. Neighbors of 'C': A (visited), E. Add E.
   * Q:
   * V: {A, B, C, D, E}
   * Visited Order: A, B, C, D, E

5. Dequeue 'D'. Neighbors of 'D': B (visited), F. Add F.
   * Q: [E, F]
   * V: {A, B, C, D, E, F}
   * Visited Order: A, B, C, D, E, F

6. Dequeue 'E'. Neighbors of 'E': C (visited), F (visited). No new unvisited neighbors.
   * Q: [F]
   * V: {A, B, C, D, E, F}
   * Visited Order: A, B, C, D, E, F

7. Dequeue 'F'. Neighbors of 'F': D (visited), E (visited). No new unvisited neighbors.
   * Q:
   * V: {A, B, C, D, E, F}
   * Visited Order: A, B, C, D, E, F

* **Order of nodes visited (BFS from A): A, B, C, D, E, F**
